An Introduction to KLEE

Cristian Cadar

Department of Computing

Imperial College London

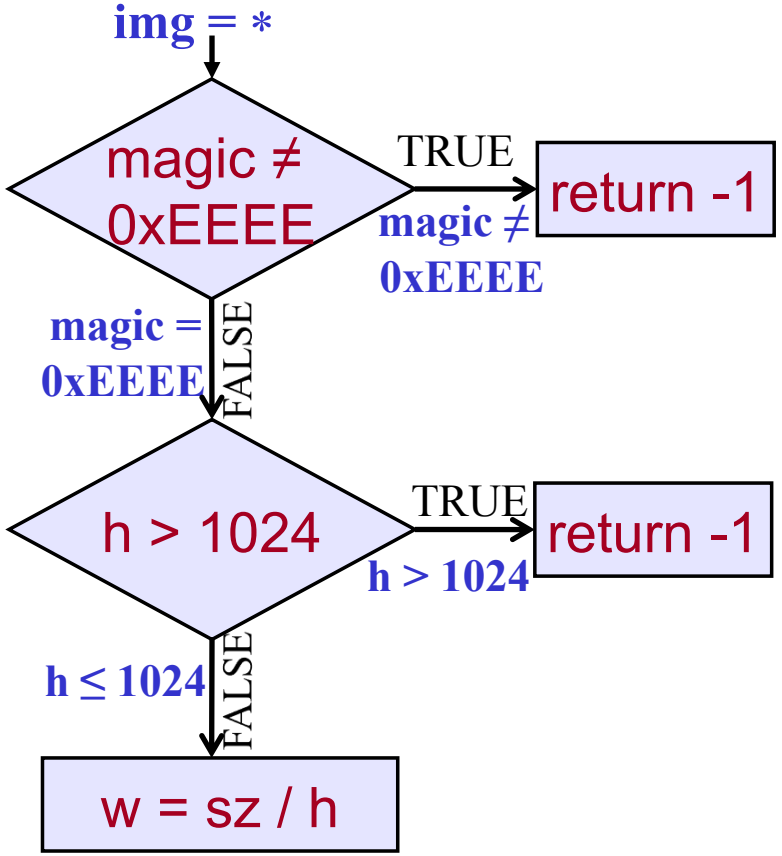
Imperial College London

Software Reliability Course
Autumn 2016

Toy Example

```
struct image_t {
     unsigned short magic;
     unsigned short h, sz;
     ...
```

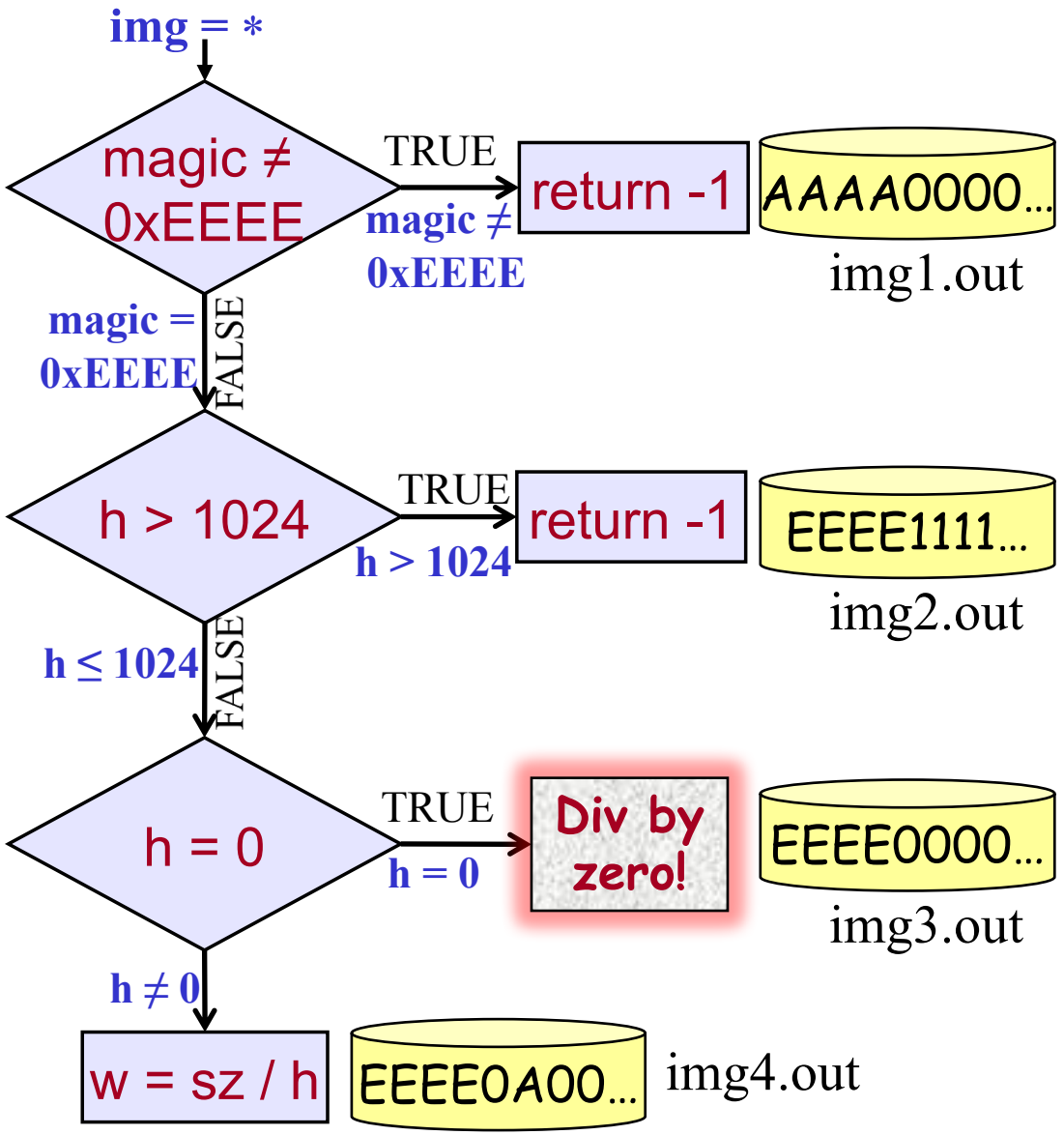
```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



Toy Example

```
struct image_t {
     unsigned short magic;
     unsigned short h, sz;
     ...
```

```
int main(int argc, char** argv) {
  ...
  image_t img = read_img(file);
  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  w = img.sz / img.h;
  ...
}
```



All-Value Checks

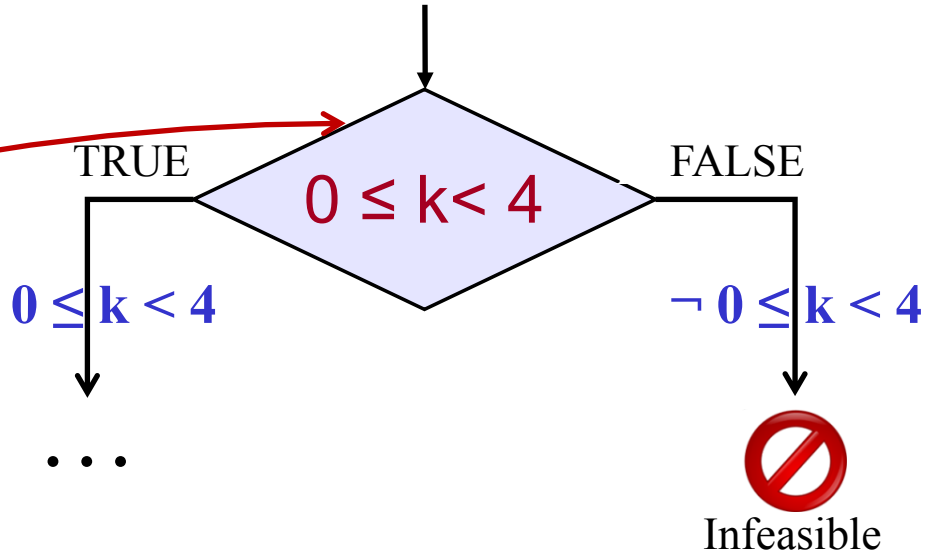
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
   int a[4] = {3, 1, 0, 4};
   k = k % 4;
   return a[a[k]];
}
```



All-Value Checks

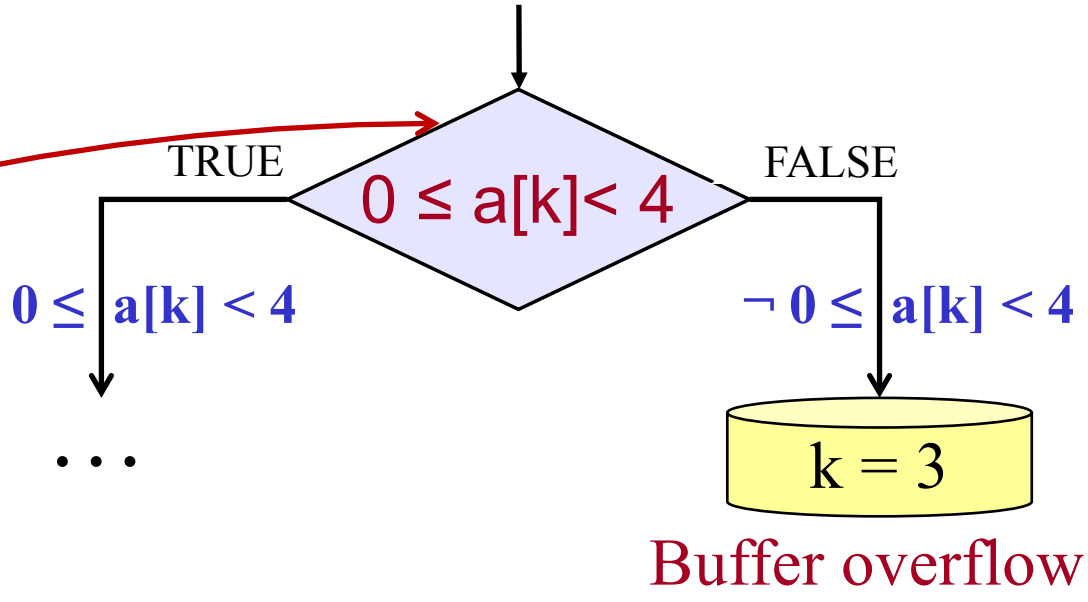
Implicit checks before each dangerous operation

- Pointer dereferences
- Array indexing
- Division/modulo operations
- Assert statements

All-value checks!

- Errors are found if **any** buggy values exist on that path!

```
int foo(unsigned k) {
   int a[4] = {3, 1, 0, 4};
   k = k % 4;
   return a[a[k]];
}
```



Buffer overflow!

KLEE

- Symbolic execution tool started as a successor to EXE
- Based on the LLVM compiler, primarily targeting C code
- Open-sourced in June 2009, now available on GitHub
- Active user base with over 300 subscribers on the mailing list and over 35 contributors listed on GitHub

Webpage: **klee.github.io**

Code: **github.com/klee**

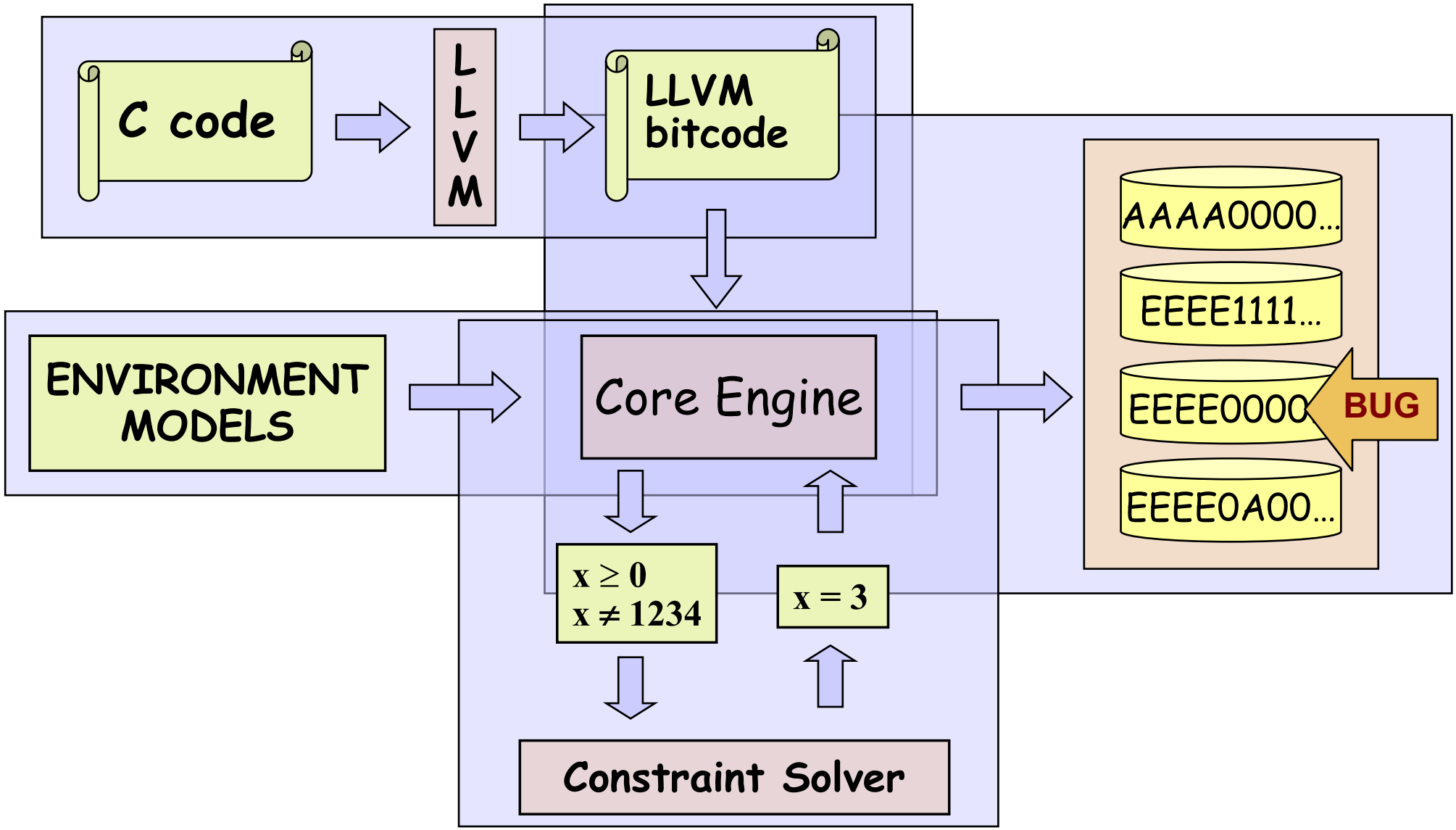
KLEE

- Extensible platform, used and extended by many groups in academia and industry, in the areas such as:
  - bug finding
  - high-coverage test input generation
  - exploit generation
  - automated debugging
  - wireless sensor networks/distributed systems
  - schedule memoization in multithreaded code
  - client-behavior verification in online gaming
  - GPU testing and verification, etc.

An incomplete list of publications and extensions available at:

klee.github.io/Publications.html

KLEE Architecture



KLEE Demo: Toy Image Viewer

```
// #include directives
struct image_t {
  unsigned short magic;
  unsigned short h, sz; // height, size
  char pixels[1018];
};
int main(int argc, char** argv) {
  struct image_t img;
  int fd = open(argv[1], O_RDONLY);
  read(fd, &img, 1024);

  if (img.magic != 0xEEEE)
    return -1;
  if (img.h > 1024)
    return -1;
  unsigned short w = img.sz / img.h;

  return w;
}
```

```
$ clang -emit-llvm -c -g image_viewer.c
$ klee --posix-runtime -write-pcs
  image_viewer.bc --sym-files 1 1024 A
...
KLEE: output directory = klee-out-1
(klee-last)
...
KLEE: ERROR: ... divide by zero
...
KLEE: done: generated tests = 4
```

KLEE Demo: Toy Image Viewer

```
$ cat klee-last/test000003.pc
...
array A-data[1024] : w32 -> w8 = symbolic
(query [
         ...
         (Eq  61166
             (ReadLSB w16 0 A-data))
         (Eq  0
             (ReadLSB w16 2 A-data))
         ...
)
```

KLEE Demo: Toy Image Viewer

```
$ klee-replay --create-files-only klee-last/test000003.ktest
[File A created]

$ xxd -g 1 -l 10 A
0000000: ee ee 00 00 00 00 00 00 00 00                    ..........

$ clang -o image_viewer image_viewer.c
[image_viewer created]

$ ./image_viewer A
Floating point exception
```

KLEE Demo: All-Values Checks

```
int foo(unsigned k) {
   int a[4] = {3, 1, 0, 4};
   k = k % 4;
   return a[a[k]];
}

int main() {
   int k;
   klee_make_symbolic(&k, sizeof(k), "k");
   return foo(k);
}
```

```
$ clang -emit-llvm -c -g all-values.c
$ klee all-values.bc
...
KLEE: ERROR: /home/klee/all-values/all-values.c:4: memory error: out of bound pointer
...
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Running KLEE inside a Docker container

Step 1: Install Docker for Linux/MacOS/Windows

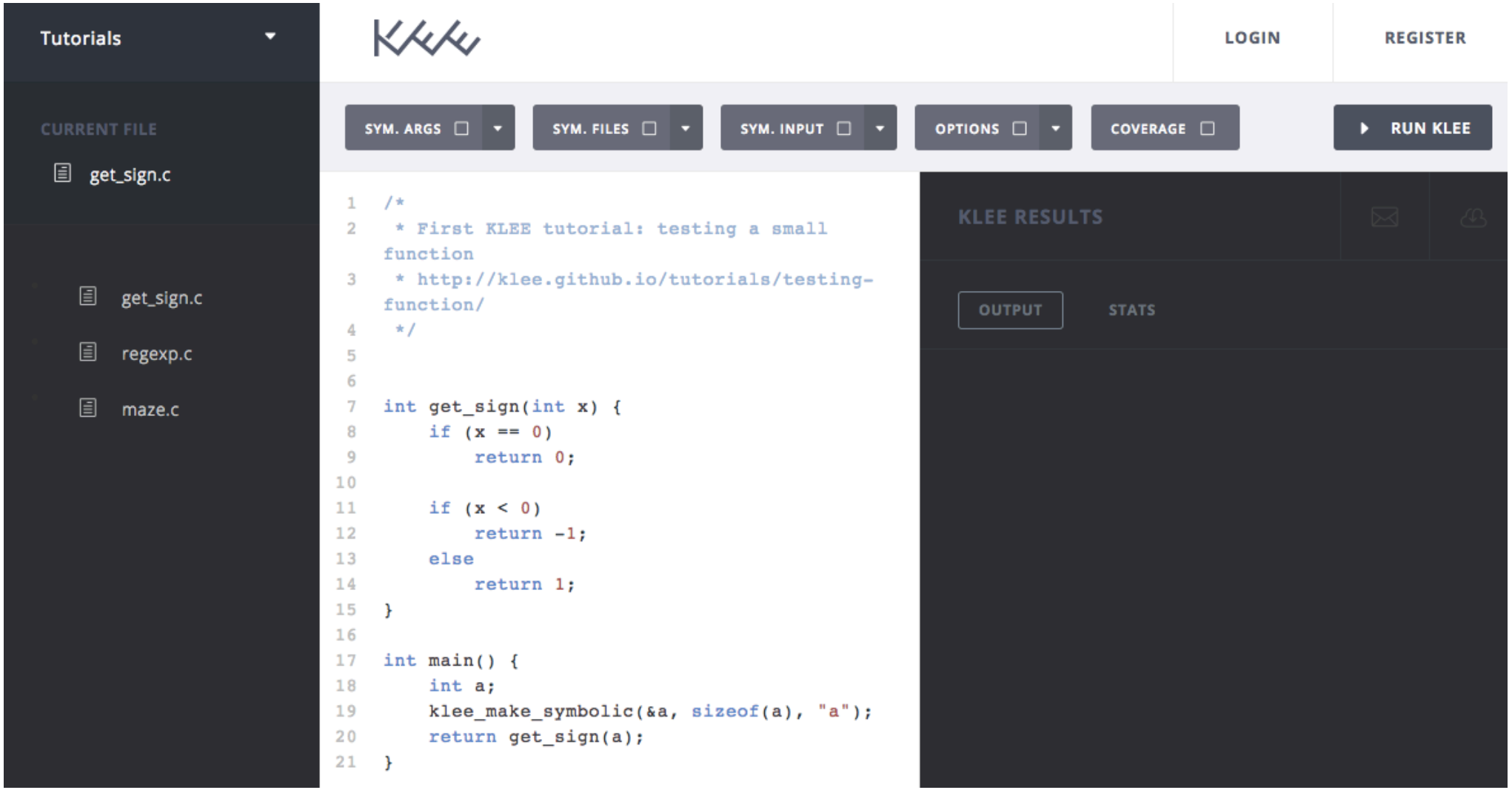
Step 2: docker pull klee/klee

Step 3: docker run --rm -ti --ulimit='stack=-1:-1' klee/klee

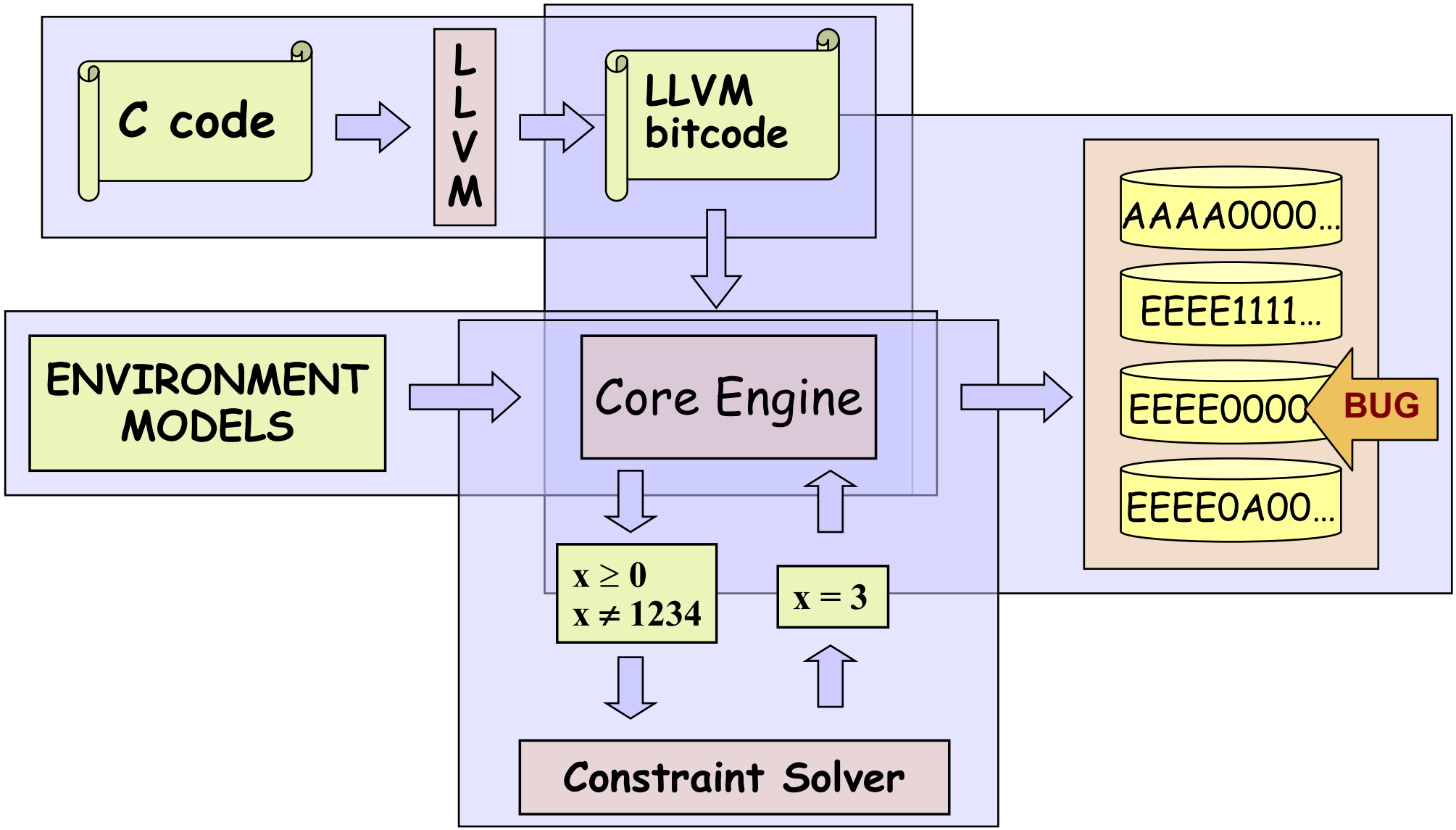
http://klee.github.io/docker/

KLEE on the Web at

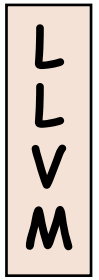
http://klee.doc.ic.ac.uk



KLEE Architecture



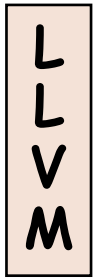
KLEE Architecture:



LLVM advantages:

- Mature framework, incorporated into commercial products by Apple, Google, Intel, etc.
- Elegant design patterns: analysis passes, visitors, etc.
- Single Static-Assignment (SSA) form with infinite registers (nice fit for symbolic execution)
- Lots of useful program analyses
- Well documented
- Several different front-ends, so KLEE could be extended to work with languages other than C

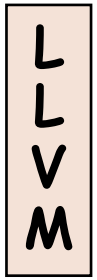
KLEE Architecture:



LLVM disadvantages

- Fast changing, not-backward compatible API!
  - KLEE is currently based on LLVM 3.4
  - But working on supporting newer versions
- Compiling to LLVM bitcode is still not trivial, but it's getting better:
  - make CC="clang –emit-llvm"
  - LLVM Gold Plugin http://llvm.org/docs/GoldPlugin.html
  - Whole-Program LLVM https://github.com/travitch/whole-program-llvm

KLEE Architecture:



KLEE runs LLVM, not C code!

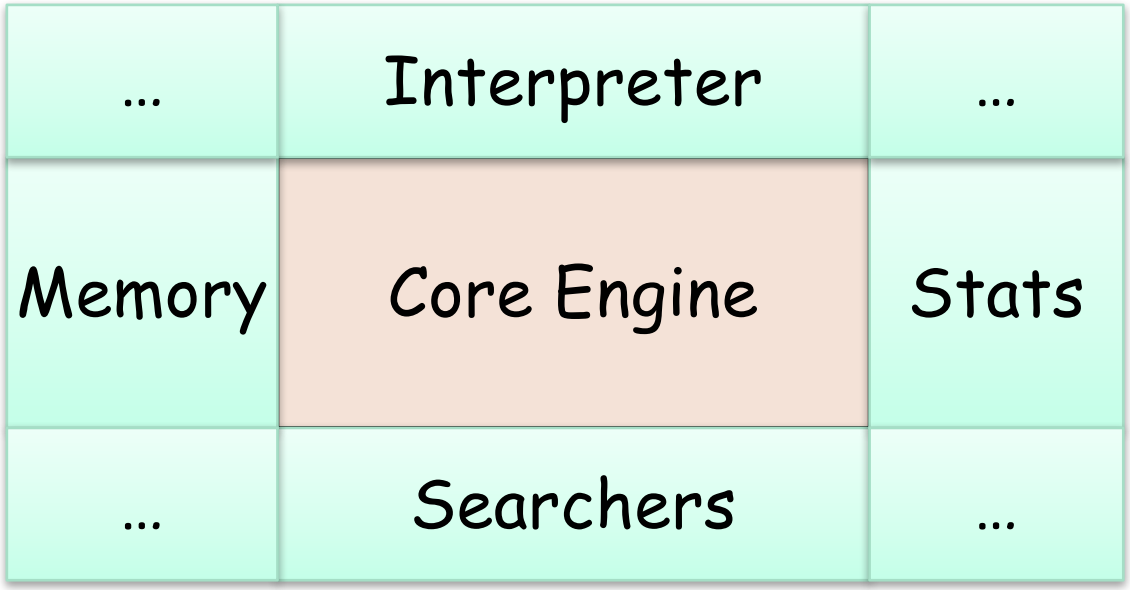
```
#include <stdio.h>
int main() {
   int x;
   klee_make_symbolic(&x, sizeof(x), "x");

   if (x > 0)
       printf("x\n");
   else printf("x\n");

   return 0;
}
```

```
$ clang -emit-llvm -c -g code.c
$ klee code.bc

...
x

KLEE: done: total instructions = 6
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

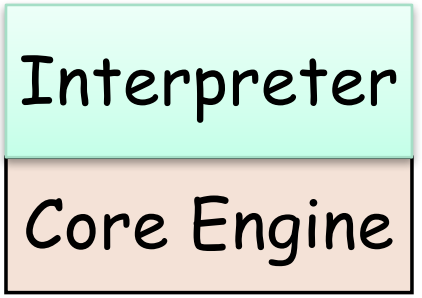
KLEE Architecture:

Core Engine

The core engine implements symbolic execution exploration.



KLEE Architecture:



- Works as a mixed concrete/symbolic interpreter for LLVM bitcode

```
Instruction *i = ki->inst;
 switch (i->getOpcode()) {
    case Instruction::Ret:
    …
    case Instruction::Br:
      // if both sides feasible, fork
      …
```

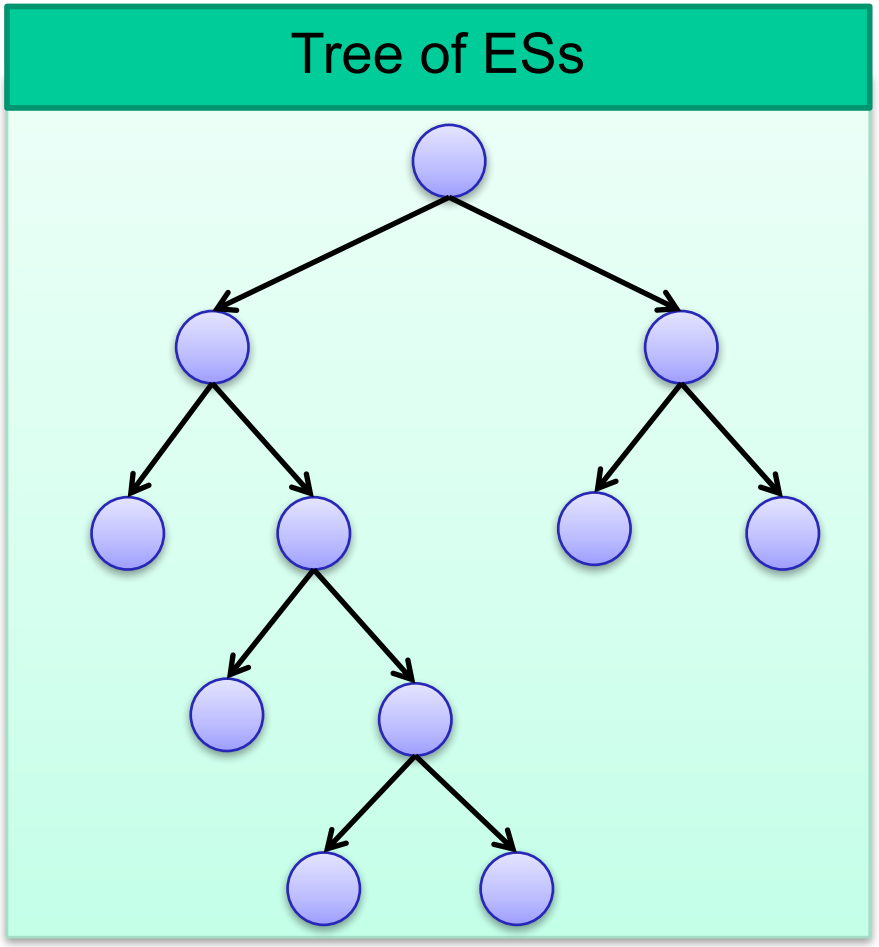
$ ./program ⟺ $ klee program.bc

Paths and Execution States

- Each path represented by an *ExecutionState*, with KLEE acting as an OS for ExecutionStates

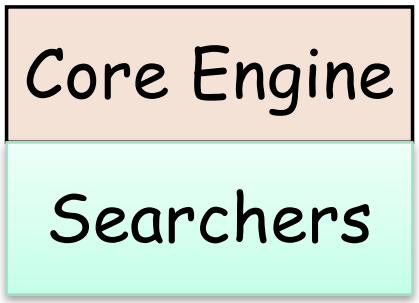
ExecutionState

- PC
- Stack
- Address space
- List of sym objects
- Path constraints
- etc.



- Fork implemented by object-level COW

Search Heuristics in KLEE



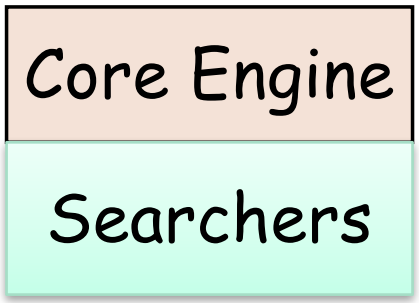
- Basic search heuristics such as BFS and DFS

```
klee --search=dfs program.bc
```

- Coverage-optimized search (`--search=nurs:md2u`)
  - Select path closest to an uncovered instruction
- Random-state search (`--search=random-state`)
  - Randomly select a pending state/path
- Random-path search (`--search=random-path`)
  - Described in the lecture
- etc.

[CCS'06, OSDI'08, ICSE'12, etc.]

Combining Search Heuristics



KLEE can also use multiple heuristics in a round-robin fashion, to protect against individual heuristics getting stuck in a local maximum.

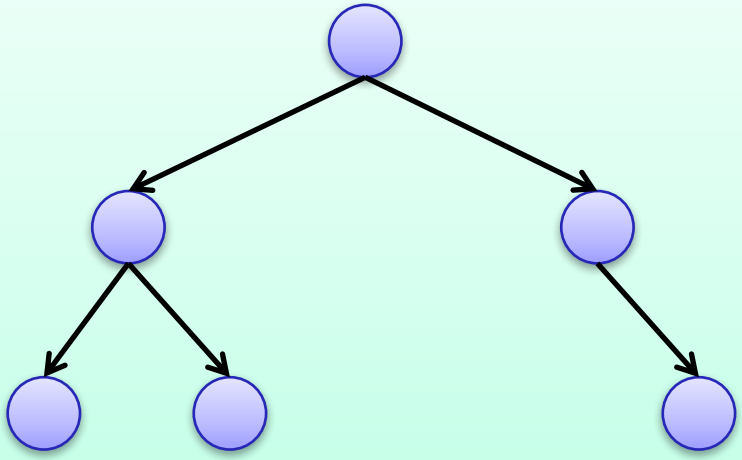
```
klee --search=nurs:md2u --search=dfs
     --search=random-path ...
```

New Search Heuristics

Core Engine

Searchers

Tree of ESs

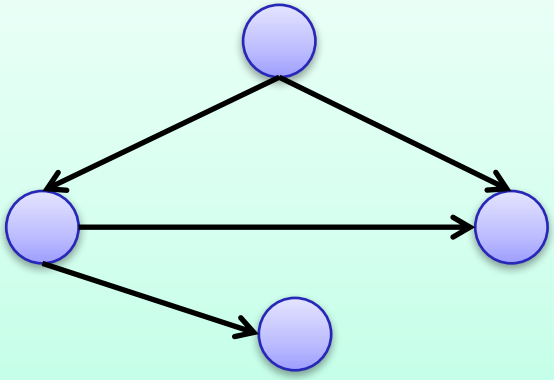


Easy to plug a new searcher by extending the Searcher class:

selectState() → ExecutionState

update(addedStates, removedStates)

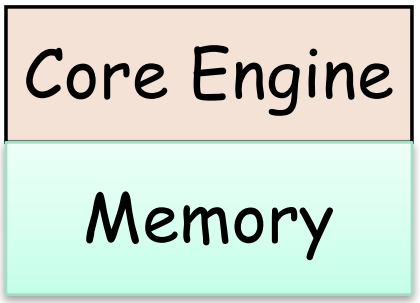
CFG



Statistics

- Solver time
- Instructions executed
- Memory consumption
- etc.

Memory Modelling



- One data type: **arrays of bitvectors (BVs)**
- Mirror the (lack of) type system in C
  - Model each memory block as an array of 8-bit BVs
  - Bind types to expressions, not bits
- We can translate all C expressions into constraints in the theory of quantifier-free BV with arrays (QF_ABV) with bit-level accuracy
  - Main exception: floating-point

Accuracy: Example

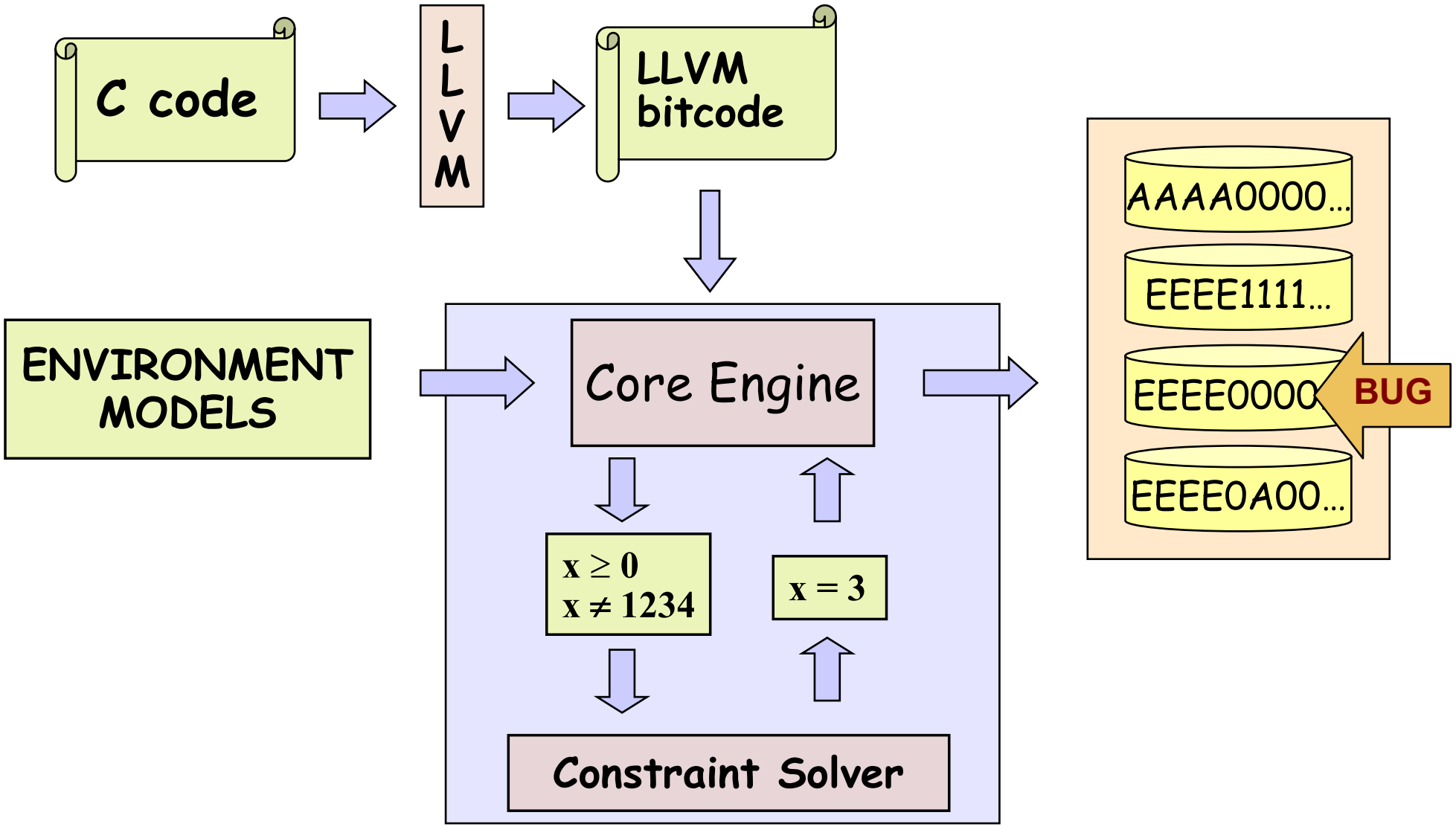
```
char buf[N]; // symbolic
struct pkt1 { char x, y, v, w; int z; } *pa = (struct pkt1*) buf;
struct pkt2 { unsigned i, j; } *pb = (struct pkt2*) buf;
if (pa[2].v < 0) { assert(pb[2].i >= 1<<23); }
```

```
buf: ARRAY BITVECTOR(32)OF BITVECTOR(8)
```

$$buf[18] <_{SIGNED} 0x00$$

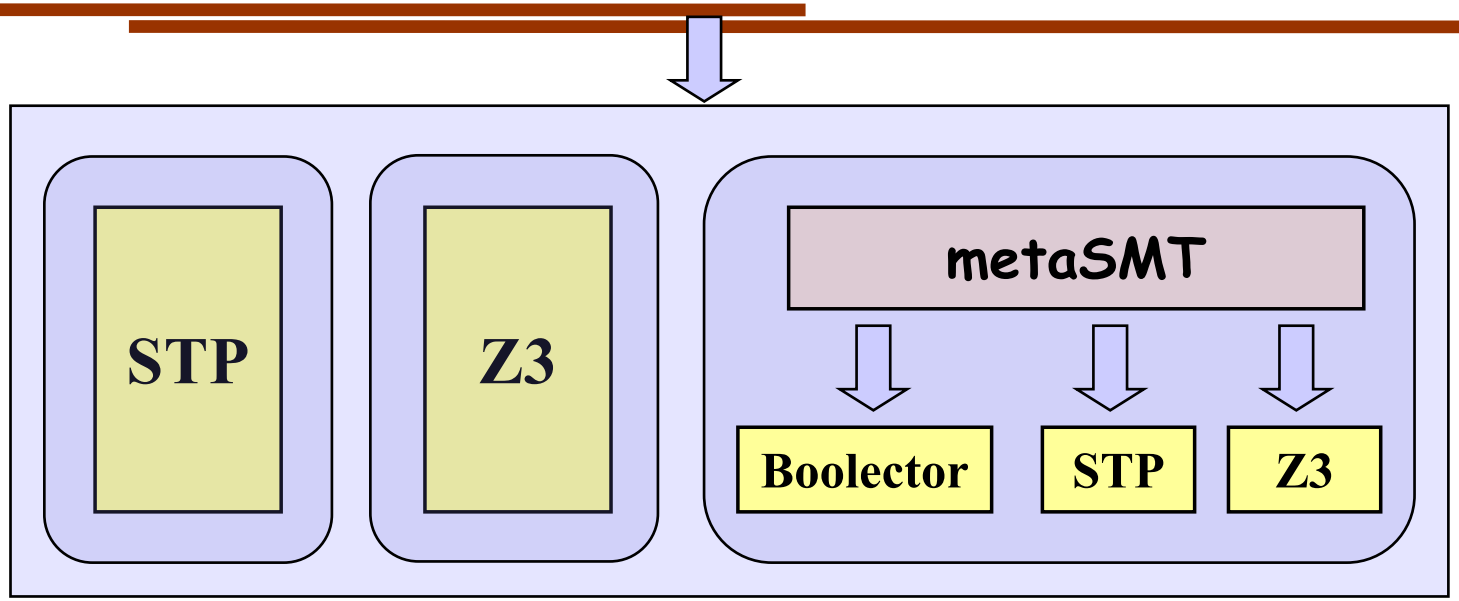
$$buf[19]@buf[18]@buf[17]@buf[16] \geq_{UNSIGNED} 0x00800000$$

KLEE Architecture



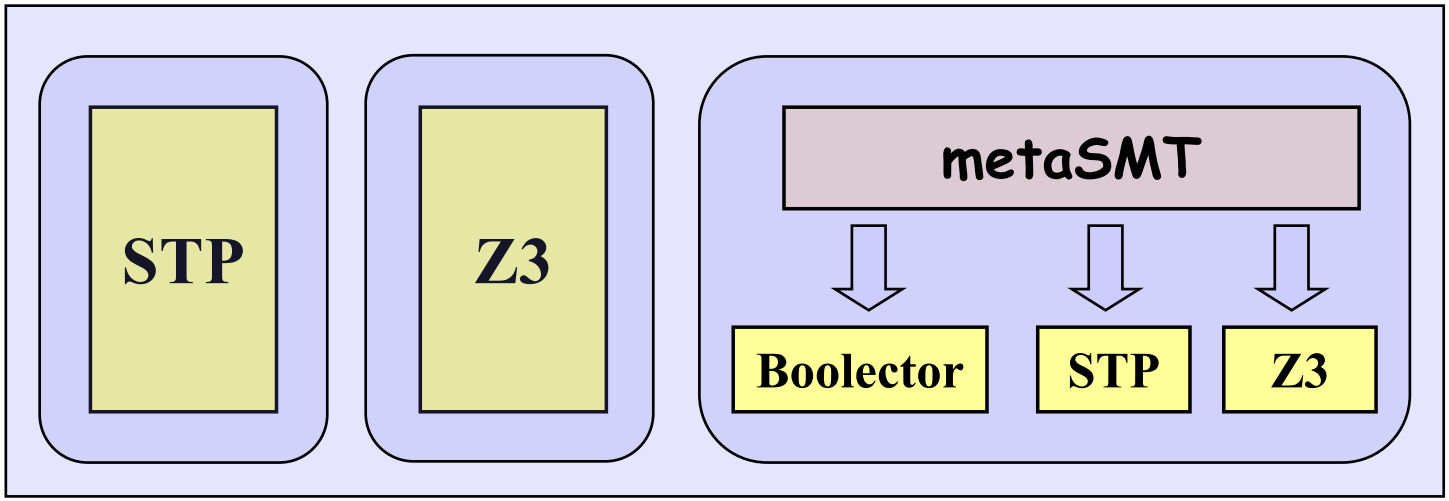
SMT Solvers
(--solver-backend=stp, z3, …)

Theory of closed quantifier-free formulas over bitvectors and arrays of bitvectors (QF_ABV)



- **STP:** Developed at Stanford. Initially targeted to, and driven by, EXE. Main solver in KLEE.
- **Z3:** Developed at Microsoft Research, integrated both natively and as part of metaSMT.
- **Boolector:** Developed at Johannes Kepler University, integrated via metaSMT.

metaSMT

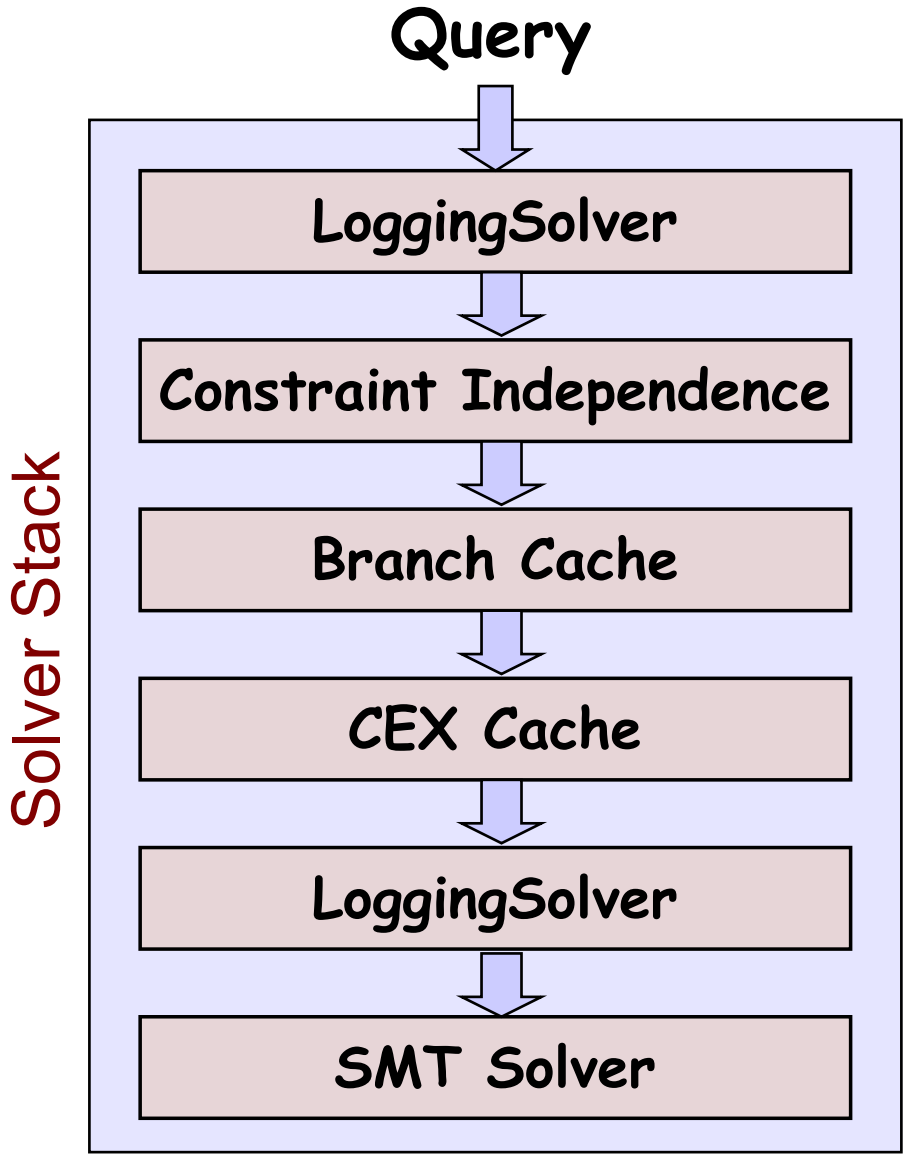


- **metaSMT** developed at University of Bremen provides a unified API for transparently using a number of SMT (and SAT) solvers
  - Avoids communication via text files, which would be too expensive
  - Small overhead: compile-time translation via metaprogramming

KLEE Architecture:

Constraint Solver

- Several high-level optimizations specific to symex
  - CEX caching, elimination of irrelevant constraints, etc.
- Implemented as a stack of solver passes
- Caching → only some queries reach the solver
- Independent **Kleaver** tool that implements this solver stack



Eliminating Irrelevant Constraints
(--use-independent-solver=true/false)

- In practice, each branch usually depends on a small number of variables

```
…
…
if (x < 10) {
    …
}
```

~~w+z > 100~~

~~2 * w – 1 < 12345~~

x + y > 10

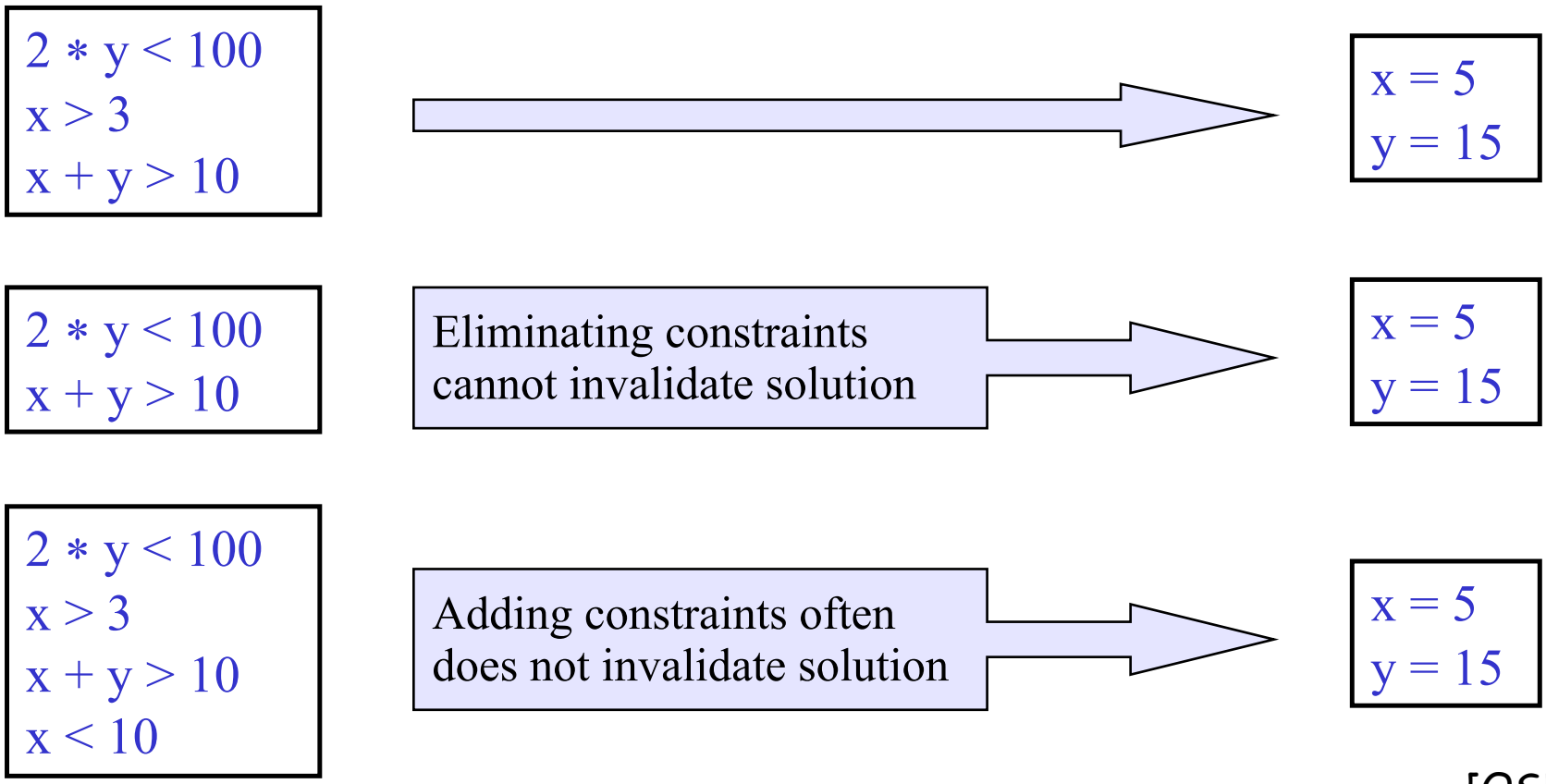
~~z & ~z = z~~

x < 10 ?

[CCS'06]

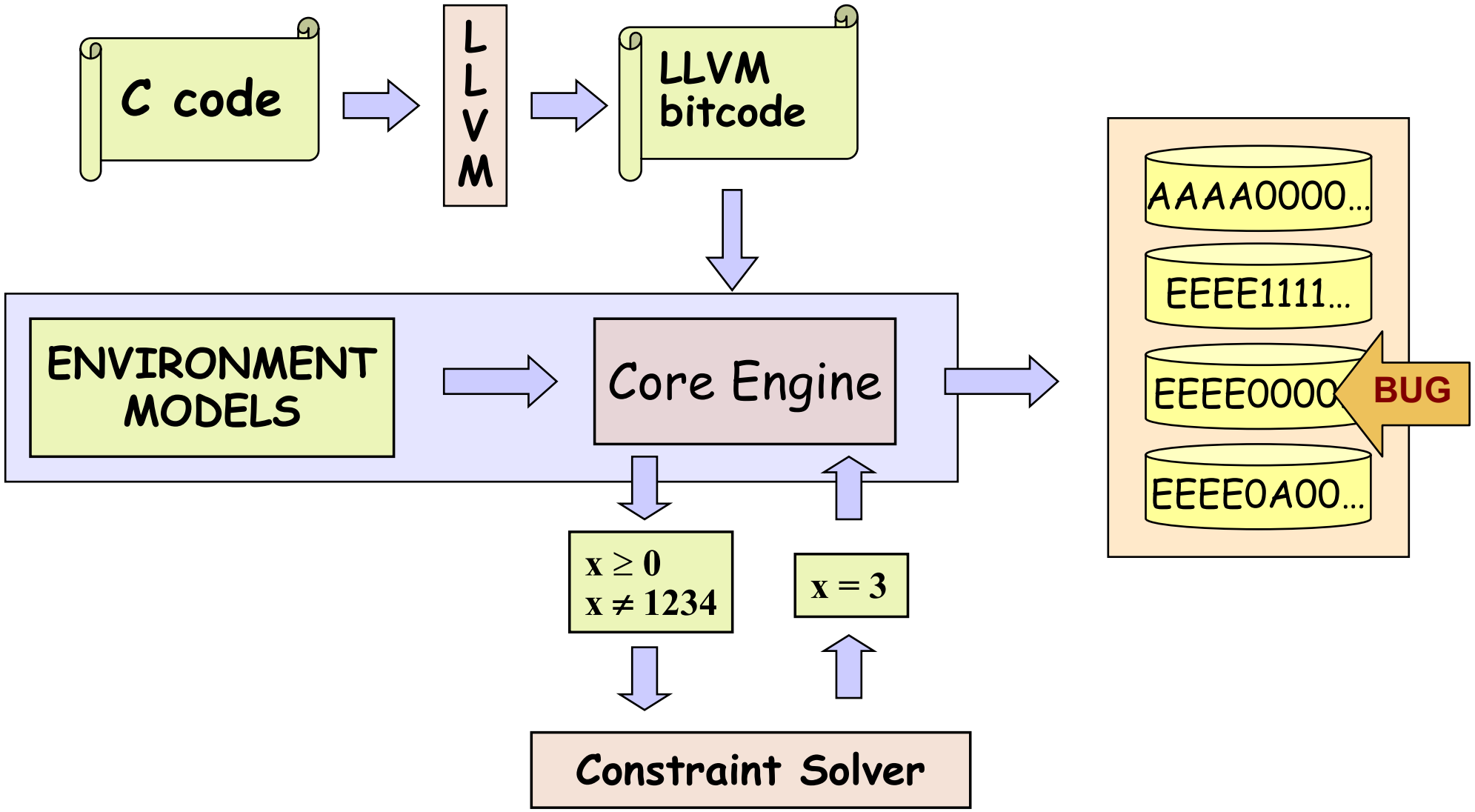
Caching Solutions
(--use-cex-cache=true/false)

- Static set of branches: lots of similar constraint sets



[OSDI'08]

KLEE Architecture



KLEE Architecture:

Environment Models

- Environment model: model for a piece of code for which source is not available
- In KLEE, the environment is mainly the OS system call API

Environmental Modeling

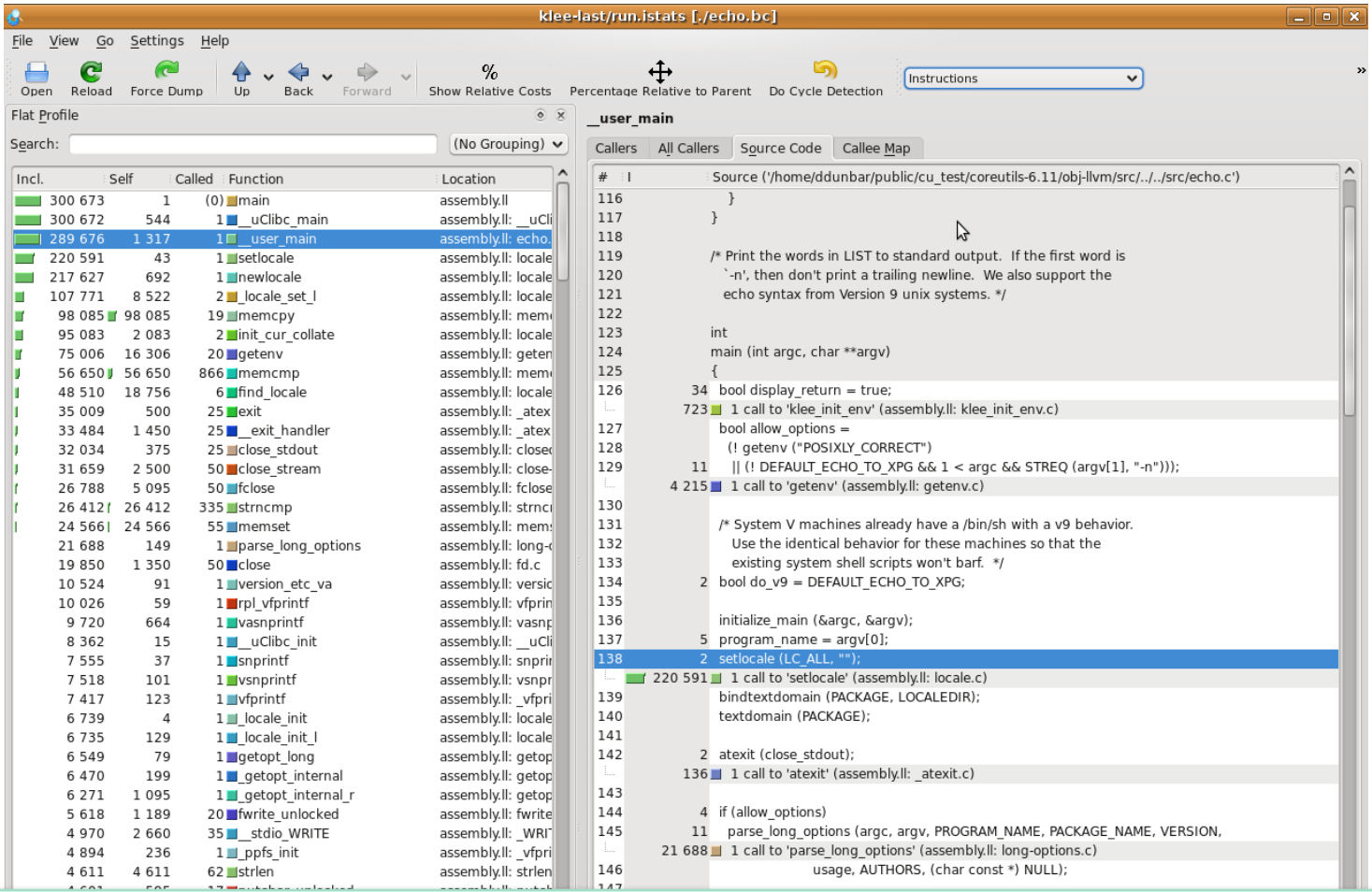
Models are plain C code, which KLEE interprets as any other code!

```
// actual implementation: ~50 LOC
ssize_t read(int fd, void *buf, size_t count) {
        klee_file_t *f = get_file(fd);
        …
        memcpy(buf, f->contents + f->off, count)
        f->off += count;
        …
```

- Users can extend/replace environment w/o any knowledge of KLEE's internals
  - Often the first part of KLEE users experiment with
- Users can choose precision
  - fail system calls? etc.
- Currently: effective support for symbolic command line arguments, files, links, pipes, ttys, environment vars

Statistics

Core Engine | Stats

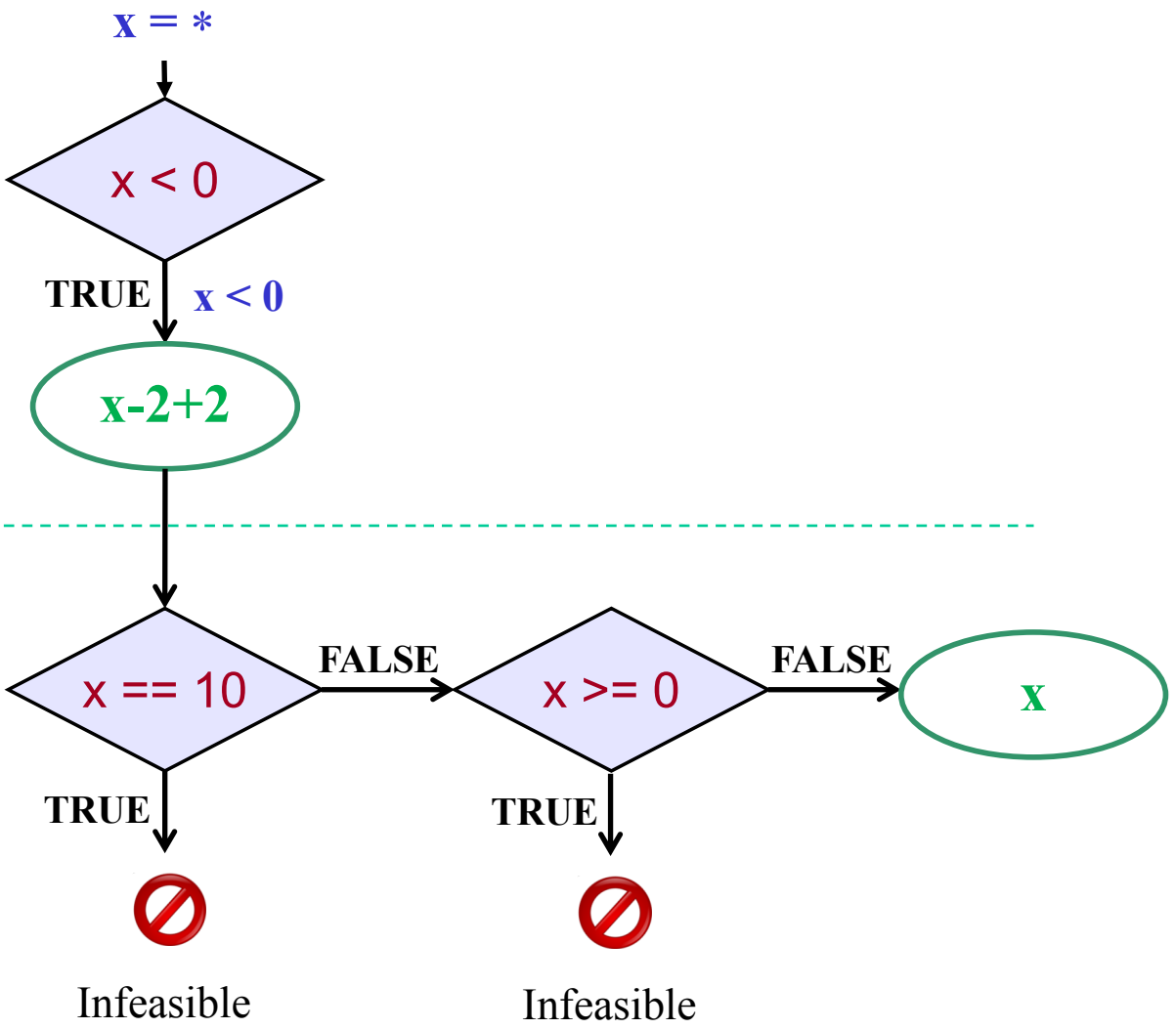


Good support for producing and visualizing a variety of statistics, associated with different entities and events

Crosschecking Two Software Versions

```
if (x < 0)
    x -= 2;
else
    if (x%2 != 0)
      x--;
return x+2;
```

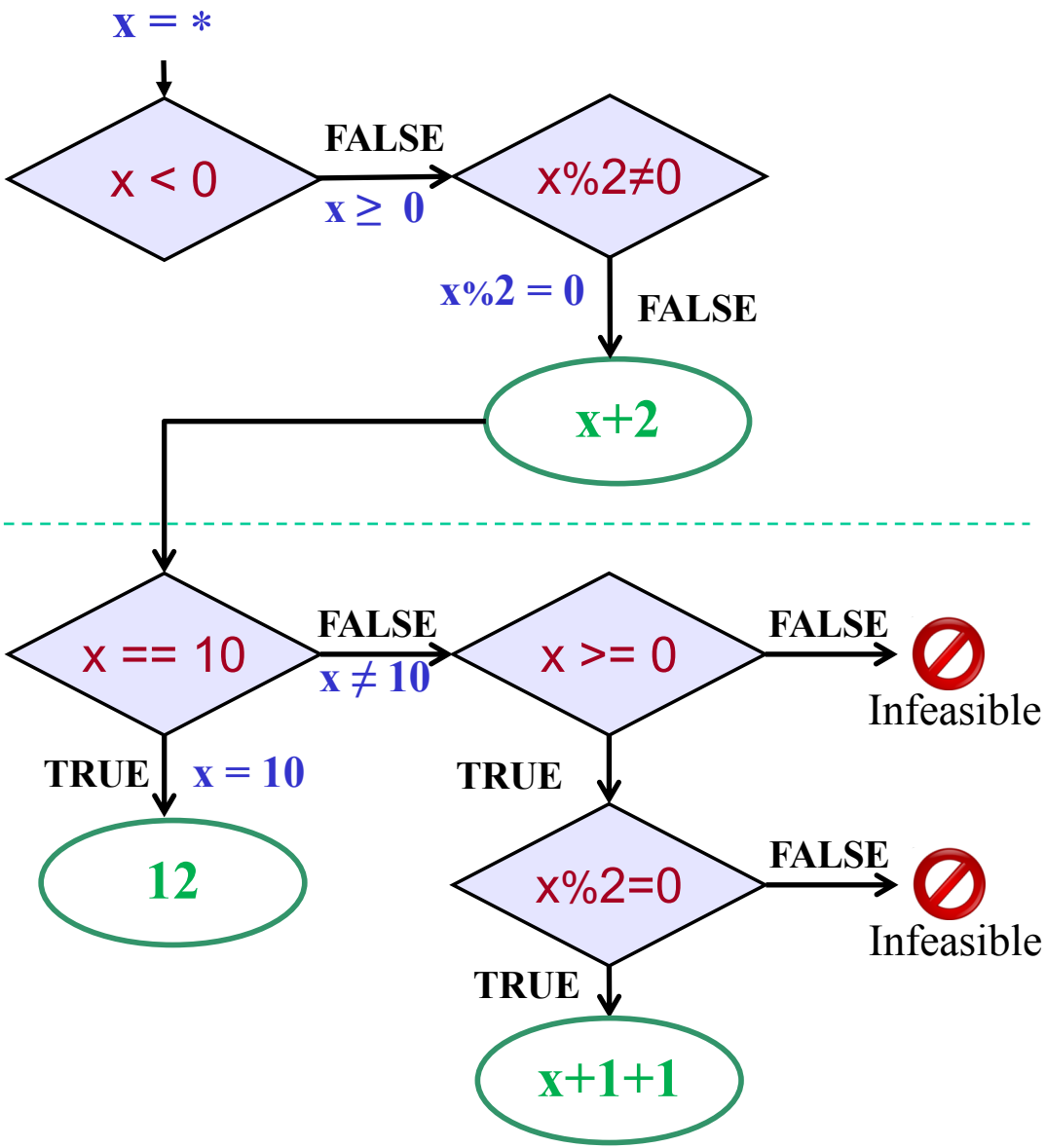
```
if (x == 10)
   return 12;

if (x >= 0) {
   if (x%2 == 0)
      x++;
   x++;
}
return x;
```



Crosschecking Two Software Versions

```
if (x < 0)
    x -= 2;
else
    if (x%2 != 0)
      x--;
return x+2;
```

```
if (x == 10)
   return 12;

if (x >= 0) {
   if (x%2 == 0)
      x++;
   x++;
}
return x;
```



KLEE: Freely Available as Open-Source

http://klee.github.io/

- Popular symbolic execution tool with an active user and developer base
- Extended in many interesting ways by several groups from academia and industry, in areas such as:
  - exploit generation
  - wireless sensor networks/distributed systems
  - automated debugging
  - client-behavior verification in online gaming
  - GPU testing and verification
  - etc. etc.